SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Automatic Benchmark Profiling through Advanced Workflow-based Trace Analysis*

Alexis Martin[3,1,2], Vania Marangozova-Martin[2,1]

[1] *CNRS, LIG, F-38000 Grenoble, France*
[2] *Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France*
[3] *Inria*

SUMMARY

Benchmarking has proven to be crucial for the investigation of the behavior and performances of a system. However, the choice of relevant benchmarks still remains a challenge. To help the process of comparing and choosing among benchmarks, we propose a solution for automatic benchmark profiling. It computes unified benchmark profiles reflecting benchmarks' duration, function repartition, stability, CPU efficiency, parallelization and memory usage. Our approach identifies the needed system information for profile computation and collects it from execution traces captured without benchmark code modifications. It structures profile computation as a reproducible workflow for automatic trace analysis which efficiently manages important trace volumes. In this paper we report on the design and the implementation of our approach which involves the collection and analysis of about 500GB of trace data coming from two different platforms (a x86 desktop machine and the Juno SoC board). The computed benchmark profiles provide valuable insights about benchmarks' behavior and help compare different benchmarks on the same platform, as well as the behavior of the same benchmark on different platforms. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: benchmarking, trace analysis, workflow, reproducibility

1. INTRODUCTION

System performance is a major preoccupation during system design and implementation. Even if some performance aspects may be guaranteed by design using formal methods [1], all systems undergo a testing phase during which their execution is evaluated. The evaluation typically quantifies performance metrics or checks behavior correction in a set of use cases. In many cases, system performance evaluation does not only consider absolute measures for performance metrics but is completed by *benchmarks*. The point is to use well-known and accepted test programs to compare the target system against competing solutions.

Existing benchmarks are numerous and target different application domains, different platforms and different types of evaluation. There are benchmarks for MPI applications [2, 3], for mobile devices [4], for web servers [5, 6] and for the Linux operating system [7] to name a few.

Constructing a benchmark is a difficult task [8] as it needs to capture relevant system behaviors, under realistic workloads and provide interesting performance metrics. This is why benchmarks

*This work has been funded by the SoC-Trace FUI project http://soc-trace.minalogic.net
*Correspondence to: Vania.Marangozova-Martin@imag.fr, ERODS team, LIG - Bâtiment IMAG - CS 40700 - 38058 GRENOBLE CEDEX 9

evolve with the maturation of a given application domain and new benchmarks appear as new system features need to be put forward. Developers frequently find themselves confronted with the challenge of choosing *the right* benchmark among the numerous available. To do so, they need to understand under which conditions the benchmark is applicable, what system characteristics it tests, how its different parameters should be configured and how to interpret the results. In most cases, the choice naturally goes to the most popular benchmarks. Unfortunately, these may not suitable for all use cases and an incorrect usage may lead to irrelevant results [9, 10, 11, 12].

To reveal benchmarks' internals and thus enhance the benchmarking process, we propose a solution for automatic profiling of benchmarks. The profiles characterize the runtime behavior of benchmarks and are computed from information contained in benchmarks' execution traces. The profile computation is structured as a deterministic and reproducible trace analysis workflow that can easily be applied to traces coming from different benchmarks or from different executions of the same benchmark. The resulting profiles are expressed in terms of well-defined metrics that help benchmark comparison and guide developers' choices.

The contributions in this paper can be summarized as follows:

- *Definition of unified profiles for benchmarks.*
  We define profiles in terms of execution duration, function repartition, stability, CPU efficiency, parallelization and memory usage. These are standard features and can be easily understood and interpreted by developers.
- *Definition of the tools needed to compute the profiles.*
  We structure the profile computation as a reproducible and generic workflow. It explicitly specifies and automatically executes the chain of data analysis and transformation. It uses a generic trace representation which allows for support of heterogenous trace formats. It is implemented using our SWAT prototype [13, 14] which provides parallel and streaming features for big data processing workflows. The final workflow is automatic and may be easily applied to different benchmarks or to different configurations of the same benchmark.
- *Definition of the data needed for profile computation.*
  We use system tracing and extract useful data in application-agnostic manner. Tracing takes advantage of standard system interfaces and does not modify or need the application source code.
- *Profiling of the Phoronix benchmarks.*
  We use our solution to profile some of the most popular benchmarks of the Phoronix Test Suite [15]. We compare benchmarks using multiple runs on different embedded and desktop platforms.

The paper is organized as follows. Section 2 positions our work. Section 3 details the benchmark profiling process. Section 4 describes its application to the Phoronix Test Suite and shows how profiles may be used to reason and choose among benchmarks. Section 5 concludes and discusses research perspectives.

2. RELATED WORK

Classical benchmark-oriented efforts focus on the design of *relevant* benchmarks. Their goal is to capture the typical behavior and measure specific performance metrics of a given system under well-defined workloads. DBench [16], for example, generates IO-intensive workloads and measures filesystem throughput. The Linpack benchmark [17] solves a dense system of linear equations to characterize peak performance in high performance computing (HPC) systems. The TPC-C benchmark [18] characterizes transactions' execution rate and pricing in transactional systems. These are just a few examples among the numerous benchmarks that have emerged in the huge exploration space of performance metrics and execution configurations.

To help the interpretation of benchmark results and facilitate the comparison among different systems, there are important efforts towards benchmark *standardisation*. A few examples are

DOI: 10.1002/spe

SPEC's [19], TCP [18] and DBench [20] benchmarks. SPEC's benchmarks test performance and power efficiency in contexts including virtualized datacenters, Java servers and HPC systems. TPC benchmarks focus on transaction processing and database performance. The DBench framework defines dependability benchmarks. While such initiatives do aim at defining reference benchmarks, their system coverage is not always disjoint and brings the issue of benchmark selection. OpenMP benchmarks, for instance, are proposed by SPEC, NASA [21] and the University of Edinbourgh [22]. To choose among benchmarks and to properly benchmark a system, one would require expert knowledge of the benchmark's specifics. The descriptions of benchmarks' functionality, however, are typically quite short. The details may be found only by investigating the benchmarks' source code which is not always freely available.

Recent benchmark-oriented efforts [15, 23, 4] focus on providing a *complete*, *portable* and *easy-to-use* set of benchmarks. The goal is to cover different performance aspects, support different platforms and be able to automatically download, install and execute benchmarks. The description of benchmarks' specifics, however, is still not a priority. There is no detailed information about benchmarks' functionality and benchmark classification is *ad hoc*. The lack of clear benchmark specifications is a main motivation for our work which aims at revealing benchmarks' operation and provides criteria for benchmark comparison.

As the final goal of our proposal is to identify relevant and representative benchmarks, it is related to the work presented in [24]. The authors use simulation to obtain source-code-related statistics reflecting the type and number of executed instructions, the branching behavior of the control flow, as well as locality characteristics. They gather statistics about different benchmark suites, treat them with cluster-based methods and identify a minimal representative benchmark set. We propose a complementary solution which considers execution-related metrics and real-world settings. It applies to binary programs and, most of all, automates the analysis.

Our proposal can be seen as a profiling tool for benchmarks. However, existing profiling tools usually provide low-level information on a particular system aspect and are system dependent. Typical profilers such as gprof [25], OProfile [26] and PAPI [27] provide *predefined* statistics about execution events. gprof intercepts the call graph of an application and provides information on the execution time of its different parts. OProfile focuses on the hardware and system-level events of Linux running codes. Finally, PAPI [27] defines a portable application interface for accessing hardware performance counters. The Valgrind framework [28] pushes the profiling effort further as, in addition to providing statistics about memory and multi-threading aspects, it allows for error detection. Glprof [29] and MemProf [30] elevate the profilers' abstraction level as they do not track simple memory accesses but consider memory accesses to applications' data structures. Our proposal targets a multifaceted macroscopic vision of benchmarks that cannot be obtained with existing tools. It is applicable to all types of benchmarks and on different platforms. Moreover, as it exploits execution traces, our solution opens the way to benchmark debugging which is not possible with a profiling approach.

To compute a profile for a benchmark, we propose to extract the needed information from execution traces. The trace analysis we provide goes beyond the current state of the art as most existing tools are system and format specific and limit themselves to time-chart visualizations and basic statistics [31, 32]. STWorkbench [33], for example, is proposed in the domain of embedded systems and uses traces in the KPTrace [34] format. Its features include a time-chart visualisation of the trace and statistics computations about the execution time and the execution events. TraceCompas [35] provides similar functionality but for CTF [36] traces in the Linux context. Another example is Scalasca [37] which works with OTF2 [38] traces and identifies communication and synchronization patterns of parallel systems. In our work, trace analysis is based on generic data representation of traces and thus may be applied to execution traces (hence benchmarks) from different systems. It is not organized as a set of predefined and thus limited treatments but may be configured and enriched to better respond to the user needs. To do so, it is structured in terms of a deterministic workflow that allows for automation and reproducibility of the analysis process.

DOI: 10.1002/spe

Our benchmark profiling approach is instantiated in the context of LTTng-obtained [39] CTF traces. As such, it is directly related to a precursor work described in [40]. Our solution enriches the obtained execution-related information and, more importantly, automates the analysis.

Concerning workflow-oriented tools, existing solutions mostly focus on the aspects of formal specification, automation and reproducibility of computations [41, 42]. CAT [43], for example, allows for composition of web services and focuses on workflow validation. Pegasus [44] allows for abstract definition of scientific workflows and for automatic mapping of these workflows onto distributed execution resources. VisTrails [45] automatically executes nested workflows while maintaining a detailed history of the workflows' execution and evolution. Our proposal specifically addresses generic trace analysis and the problem of huge traces.

3. AUTOMATIC PROFILING OF BENCHMARKS

This section presents the benchmark profiles (Section 3.1), the LTTng tool used for benchmark tracing (Section 3.2), the needed data for profile computation (Section 3.3), the specifics of the computation (Section 3.4) and its workflow automation (Section 3.5).

3.1. Benchmark Profiles Definition

The profile considered for a benchmark is independent of its semantics and is composed of the following features:

- *Duration.* This metric gives the time needed to run the benchmark. It allows developers to estimate the time-cost of a benchmarking process and to choose between short and long-running benchmarks.
- *CPU Utilization.* This metric characterizes the way a benchmark runs on the target system's available processors. It gives information about the CPU usage, as well as about the benchmark's parallelization. It helps discovering whether the benchmark may benefit from the presence of multiple processors or, on the contrary, is sequential. If the benchmark is parallelized, this feature provides information about its load balancing among processors.
- *Kernel vs User Time.* This metric gives the distribution of the benchmark execution time between the benchmark-specific (user) and kernel operations. This gives initial information on the parts of the system that are stressed by the benchmark.
- *Benchmark Type.* The type of a benchmark is defined by the part of the system which is stressed during the benchmarking process. Namely, we distinguish between benchmarks that stress the processor (CPU-intensive), the memory (memory-intensive), the system, the disk, the network or the graphical devices. The motivation behind this classification is that it is application-agnostic and may be applied to all kinds of benchmarks.
- *Memory Usage.* This part of the profile provides information about the memory footprint of the benchmark, as well as the memory allocation variations.
- *Stability.* This metric reflects the execution determinism of a benchmark, namely the possible variations of the above metrics across multiple runs.

3.2. LTTng Tracing

The profile computation needs detailed data about the benchmark's execution. It needs timing information about the global execution and about fine-grained operations. It also needs information about the number, the type and the scheduling of the different execution events.

To collect this data, we decide to use tracing which provides a historical log containing timestamped execution information. We propose to use LTTng [39, 46, 47], the *de facto* standard for tracing Linux systems, which comes with minimal system intrusion [48, 49].

LTTng is distributed as an open source toolkit including tracing modules and a command-line tool for trace control. It gives access to hardware counters and provides tracing of both kernel and user-level operations.

DOI: 10.1002/spe

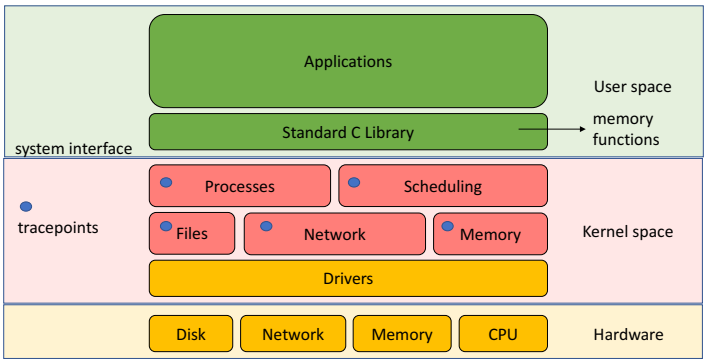


Figure 1. System tracing

LTTng's kernel tracing relies on *tracepoints* to intercept typical I/O, process management and IRQ management activities. During a tracing session, only a subset of these tracepoints may be activated in which case predefined *probe* functions are triggered to record information about the intercepted events. The LTTng user is given the possibility to define new tracepoints, as well as to replace the predefined probe functions. The recorded event data typically includes a timestamp (when the event occurred), a type (what event occurred) and a specific payload containing more detailed information about the event.

At user-level, LTTng provides the mechanisms for defining, activating and tracing user-defined tracepoints for C, C++ and Java applications.

At the hardware level, LTTng gives access to multiple hardware counters through the *perf* [50] tool (LTTng's perf contexts). It can, for example, obtain the values for CPU cycles, branch instructions or cache misses. The counters' values are added as additional data fields to traced events' information.

3.3. Initial Profile Data

As benchmarks are highly heterogenous, we propose a generic tracing solution. It considers three levels of standard interfaces reflecting user, kernel and hardware operation (cf. Figure 1).

At the user level, we focus on the interface that is commonly used by benchmarks, namely the standard C library (*libc*). By redefining the `LD_PRELOAD` environment variable and overloading the *libc* functions, it is easy to obtain the information about the memory management functions. We intercept the `malloc`, `calloc`, `realloc` and `free` functions and use the available data to compute the benchmarks' memory profiles. For benchmarks that do not use the *libc* interposition layer, such as statically built programs or programs with *setuid* privileges [51], an appropriate interception mechanism is to be used.

At the kernel level, we use the predefined LTTng *tracepoints* to intercept context switches, interruptions, system calls, memory management and I/O. For example, we track `sched` events to deduce scheduling and thus CPU usage decisions. Another example are system calls (`syscall` events) which reveal what system services the benchmark explicitly requires. The detail of how the tracepoint information is used for profile computation is given in the next section.

At the hardware level, we use the LTTng's perf contexts to access the `Instruction`, `L1-dcache-loads` and `L1-dcache-stores` counters to quantify CPU- and memory-related operations.

3.4. Profile Computation

To compute the different metrics that compose a benchmark profile we proceed as follows.

3.4.1. Benchmark Processes. When LTTng tracing is enabled, the captured trace contains the events related to the execution of the whole system and not only to the target benchmark. If, for

DOI: 10.1002/spe

example, there are some system deamons running, the final trace will also reflect their execution. As a consequence, to profile a benchmark using the LTTng system trace, we need to filter out unrelated events.

To identify the events related to the benchmark, we need to know which process produced which execution event. This information is not directly provided in the LTTng execution traces as we have chosen to minimize the tracing overhead by not attaching the corresponding `PID` to each execution event.

To discover the involved processes and delimit their activity, we focus on the process management events. In particular, we use the fact that the `sched_switch` event contains information about the previous and the new process to execute on a CPU. Besides discovering the `PID`s of the two processes, we can deduce that the events that happened between a `sched_process_fork` event and a first `sched_process_switch` event are part of the execution of the first process. The events between two consecutive `sched_process_fork` events belong to the newly scheduled process.

3.4.2. Duration and Kernel vs. User Time. We report on the perceived benchmark duration which is the time between the start and the end of the benchmark execution. This is typically the result provided by the `time` function. To compute it from the LTTng trace, we work with the events marking the start (`sched_process_fork` and `sched_process_exec`) and the end (`sched_process_exit`) of a process. The computed duration reflects the sum of the benchmark's activity time during which the CPU is occupied and idle time.

To compute a benchmark's activity time, we consider the periods during which its processes are active (scheduled), as discovered during the process tree computation. The events indicating the start and the end of system calls (`syscall_entry` and `syscall_exit`) help us establish what part of the activity time is spent in kernel mode. The remaining part is time spent in user mode.

The difference between a process' duration and its activity time gives us the time during which the process is inactive. Our current computation is yet to be enriched in order to distinguish between inactivity periods due to scheduling and off-CPU periods due to blocking.

3.4.3. CPU Utilization. LTTng tags all traced events with the CPU which triggered them. To characterize CPU utilization, we combine this information with the obtained knowledge about the benchmark's activity periods. The simple fact that a benchmark process is scheduled on a given CPU provides us with the information that this particular CPU has been used. To characterize its usage, we consider the sum of periods during which the CPU is occupied by a benchmark process. Correlating this to the benchmark execution duration gives the time during which the CPU has *not* been used by the benchmark i.e. its *idle* time. Finally, considering the usage of all available CPUs reflects whether and how the benchmark is parallelized and load balanced.

3.4.4. Benchmark Operation. To characterize the benchmark operation at the kernel level and understand which part of the system is tested, we have analyzed and classified the Linux kernel events as shown in Table I. We identify events related to CPU activity (*Processor* family), to memory operations (*Memory* family), *System* events, *Disk*-related events, *Network* events and graphical events (*Graphics*). The `mm_page_alloc` and `mm_page_free`, for example, are clearly memory-related events, while `power_cpu_idle` and `htimer_expire` are related to the CPU.

Family	Tracepoints
Processor	`timer_*; hrtimer_*; timer_*; power_*; irq_*; softer_*;`
Memory	`kemem_*; mm_*`
System	`workqueue_*; signal_*; sched_*; module_*;rpm_*; lttng_*; rcp_*; regulator_*; remap_*;recache_*; random_*; console_*;gpio_*;`
Graphics	`v4l2_*; snd_*;`
Disk	`scsi_*; jbd2_*; block_*;`
Network	`udp_*; rpc_*; sock_*; skb_*; net_*; netif_*;napi_*;`

Table I. Classification of LTTng kernel tracepoints

DOI: 10.1002/spe

To characterize a benchmark globally and thus consider both kernel and user level operation, we investigate its behavior in terms of CPU and memory-related activity. As all data accesses go through the L1 cache, we use the `L1-dcache-loads` and `L1-dcache-stores` hardware counters to obtain the total number of data related instructions. To get the number of computation related instructions, we use the `Instruction` counter and compute the difference `Instruction` − (`L1-dcache-stores` + `L1-dcache-stores`).

3.4.5. Memory Usage. As indicated in 3.3, we intercept and trace the calls to the *libc* library. The corresponding tracing events contain the name of the memory-related function, the size of the involved memory and the corresponding memory pointer. Following the timeline of the trace, we discover when memory is allocated (`malloc` and `calloc` events), reallocated (`realloc` events) and freed (`free` events or automatic release at the end of a process indicated by `sched_process_exit`).

3.4.6. Stability. For a benchmark to provide relevant performance results about a target system, its execution should not vary in an important way from one run to another. To investigate this issue, we execute and trace benchmarks multiple times (32 times for statistically relevant results [52]). We either consider the mean and the standard deviation for simple numerical values (e.g. benchmark duration, number of traced events, CPU utilization) or use visual representations (e.g. for the memory evolution profile).

While focusing on stability, a major aspect we consider is the benchmark *score*. We investigate whether a benchmark produces the same result (*score*) when testing a target system 32 times.

3.5. Automating Computation

Our profile computation is a multi-phase process which is implemented as a *workflow* with our SWAT[‡] prototype. Proposing a *workflow*-oriented approach helps us provide for reusable, reproducible and automatic trace analysis. Indeed, using a workflow allows for a step-by-step design during which the developer explicitly defines all the different analysis treatments and their interconnections. As all trace analysis actions are reflected in the workflow, the same analysis may be (partly or fully) reused and applied to different contexts and data. Having a global description of the analysis treatments makes it possible to automate the process and save the analyst the effort of launching computations, waiting for their termination and transforming data by hand. Finally, the specification of the whole process and its automation enable reproducibility: it is possible to run the analysis later by third parties to obtain the same results.

In the following we introduce the used trace representation and the SWAT basics before focusing on the workflow for benchmark profiling.

3.5.1. Trace Representation. LTTng traces represent the execution history in terms of *punctual* events. These are events that reflect a system state change at a given point of time (the event's timestamp). During our benchmark profiling, we reorganize trace data and use three more event types, namely *states*, *links* and *values*. *States* represent time intervals (durations) corresponding to system states, computation phases, function calls, etc. *Links* represent the causal relationship between two events. It may typically concern the sending and the reception of a message. Finally, a *value* reflects the evolution of a computed numerical value throughout the execution.

The above trace representation has been extensively investigated and used in the SoC-TRACE FUI project [53] and has proven to be generic enough to reflect the contents of multiple trace formats. We have successfully exploited not only LTTng traces which are in the CTF format [36] but also parallel OTF traces [38], embedded KPTrace traces [54] and multimedia gstreamer traces [55].

In the profile computation workflow, we use, among others, *states* to represent the activity periods of processes, *links* for context switches and *values* to follow memory consumption.

[‡]SWAT is our abbreviation for *Système de Workflow pour l'Analyse de Traces*. It translates from French to "Workflow System for Trace Analysis".

DOI: 10.1002/spe

```
class Consumer: public Module{
public:
  InputPort<uint64_t, Stream> d;
  void compute();
};

Consumer::compute(){
  uint64_t e;
  uint64_t res = 0;

  while (true) {
    try {
      e = d.recv();
    } catch (end_of_data) {
      break;
    }
    res += e;
  }
}
```

```
class Producer: public Module{
public:
  OutputPort<uint64_t, Stream> n;
  void compute();
};

Producer::compute() {
  uint64_t i = 0;
  while (true) {
    n.send(i);
    i += 1;
  }
}
```

```
// creation of the workflow
Workflow w;

// instanciate the two modules
Producer p ;
Consumer c ;

// add modules into the workflow
w. add_module(&p) ;
w. add_module(&c ) ;

// link the two ports of the
modules
w.link(p.n, c.n);

// run the workflow
w. start () ;
```

Figure 2. Programming a simple SWAT workflow

3.5.2. The SWAT Prototype. The major motivation behind SWAT is that the classical store-and-analyze-later approach has prohibitive storage and computation time costs when it comes to huge traces. Indeed, as it consists in first storing the trace on a persistent storage and then loading trace data for trace analysis, manipulating important data volumes may be slow or even impossible. On one hand, huge traces as the ones we have obtained for the experiences in this paper (cf. Section 4.8) do not fit into memory. On the other, trace analysis treatments that need to go through the whole trace before outputting a result, are too long.

SWAT supports data streaming workflows. A workflow is a set of interconnected *modules* which encapsulate reusable treatments for trace analysis. The execution of modules may be parallel as it is ensured by independent control flows (*threads*). In the case of modules that have a data (producer-consumer) dependency, data is streamed from the producer to the consumer and the latter starts its execution as soon as there is available data.

The workflow programming model of SWAT follows the standard component-oriented [56] approach according to which the components of an application (the modules) have unified interfaces and are interconnected to compose the application. In SWAT's workflows, module interconnections are done using the modules' *ports* which explicitly indicate whether a module produces or consumes data. The interconnections represent data streams flowing from a producer module to one or multiple consumers.

SWAT is implemented in C++ and workflows, modules and ports are respectively represented by the `Workflow`, `Module` and `InputPort/OutputPort` classes. The `Workflow` class maintains a list of the workflow's modules and provides the methods for adding a module to the workflow (`addModule`), for interconnecting modules (`link`) and for starting the computation (`start`). The `Module` class defines a method to start a module and a virtual method `compute` to contain a specific trace analysis treatment. To develop a specific module for trace analysis, one needs to extend the `Module` class and implement the `compute` method. Finally, `InputPort` and `OutputPort` represent respectively input and output module ports. They allow for specifying them as ports exchanging single values or working with streams and define the needed communication buffers and related synchronisation mechanisms.

Figure 2 shows an example of defining a simple workflow containing two modules, a producer and a consumer. The `Consumer` defines its input port as a stream if integer values. Its behavior is defined in its `compute` function which receives and simply sums the stream values. The `Producer` generates the stream and sends the values using its output port. The two modules are added and connected in the workflow, as shown in the third piece of code. The launching of the workflow will launch the execution of both modules. They will execute in parallel except in the cases when the consumer is blocked because of unavailable data.

Workflows may also be nested: a workflow may be encapsulated and appear as a single module in a higher-level workflow (Figure 3). In this case ports of workflow modules are to be linked (mapped) to ports of the encapsulating module.

DOI: 10.1002/spe

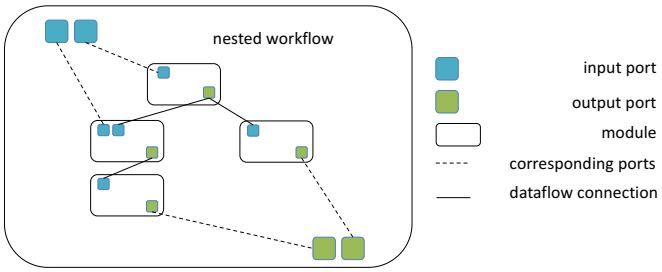


Figure 3. A SWAT workflow

The intuitive programming model of SWAT takes away the difficulties of managing the execution of a workflow and helps developers focus on the analysis specifics of their problem. The workflow we have defined for benchmark profiling is described in detail in the next section.

3.5.3. Benchmark Profiling Workflow. The global SWAT workflow for benchmark profiling is shown in Figure 4. The six modules execute in parallel and provide the following trace analysis treatments:

- `ProfileAll`. This module uses the kernel-level tracing data to compute the execution duration and to analyze both kernel execution and CPU usage. It is responsible for computing the first three metrics of the benchmark profile and provides information about the benchmark type.
- `MemoryAll`. This module is in charge of analyzing the memory usage of the benchmark.
- `PerfAll`. This module investigates whether the benchmark is CPU- or memory-intensive by quantifying the memory and CPU related operations.
- `TimeAll`, `ScoreAll` and `TraceAll`. These three modules investigate benchmark stability by respectively considering the execution durations, the benchmark scores and the traces (number of events and size) obtained during the different runs.

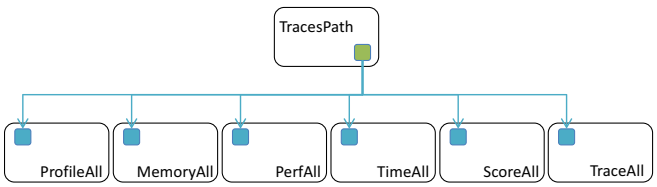


Figure 4. The benchmark profiling workflow

The code for defining the global workflow is straightforward, as shown in Figure 5. The workflow is created (line 3), the six modules are instantiated (lines 5, 9, 13, etc.), added to the workflow (lines 6, 10, 14, ...), connected to specific modules providing the paths to benchmarks' traces (lines 7, 11, 15, ...) and the analysis is launched (line 28).

Each module in the global workflow follows the same pattern: it creates multiple instances of a dedicated nested workflow which works with a single execution trace. For 32 execution benchmark runs resulting in 32 execution traces, a global module launches 32 sub-workflows and analyzes the 32 sets of results.

The `PerfAll` module, for example, uses 32 `GetPerfValues` modules, obtains two counter results per module and computes the final statistics (Figure 6).

The schematic view of `PerfAll` (Figure 6a) is easily mapped to its code (Figure 6c). The information on the number of runs (`number_of_runs`= 32) is used to instantiate and connect 32 `GetPerfValues` modules (lines 14 to 18). The memory-related and CPU-related instruction counters of each module are passed to two modules (`vts_mem_instr` and `vts_cpu_instr`,

DOI: 10.1002/spe

```
1  int main(int argc, const char * argv[]) {
2     ...
3     Workflow w (bench);
4     ...
5     GetMemoryAllRuns *mall = new GetMemoryAllRuns;
6     w.addModule(*mall);
7     w.link(bench_name->constant_value_out, mall->bench_name);
8     ...
9     GetTimeAllAnalysis *tall = new GetTimeAllAnalysis;
10    w.addModule(*tall);
11    w.link(...);
12    ...
13    GetScoreAllAnalysis *sall = new GetScoreAllAnalysis;
14    w.addModule(*sall);
15    w.link(...);
16    ...
17    GetProfileAllRuns *pall = new GetProfileAllRuns;
18    w.addModule(*pall);
19    w.link(...);
20    ...
21    GetPerfAllRuns *perfall = new GetPerfAllRuns;
23    w.addModule(*perfall);
24    ...
25    GetTraceAllAnalysis *traceall = new GetTraceAllAnalysis;
26    w.addModule(*traceall);
27    ...
28    w.start();
29    return 0;
30 }
```

Figure 5. Programming the benchmark profiling workflow

lines 9 to 12) that transform the individual values into streams. Finally, two `ComputeStats` modules are in charge of computing the statistics about the 32 runs. The `ComputeStats` module is provided by SWAT as a part of a module library for frequent operations including event counting, filtering and statistics.

If we zoom into `GetPerfValues` (Figure 6b), the module reads a trace and considers the hardware counter events to compute the numbers of computing and memory-access instructions. The trace is read by a `CTFReader` module which uses the Babeltrace interface [57] to read CTF traces and generates a corresponding stream of trace events.

4. PROFILING THE PHORONIX TEST SUITE

In this section, we first introduce the Phoronix Test Suite and our experimental setup. Thereafter, we detail the results obtained for the Phoronix benchmarks and show how these results may be used as guidelines for system benchmarking analysts.

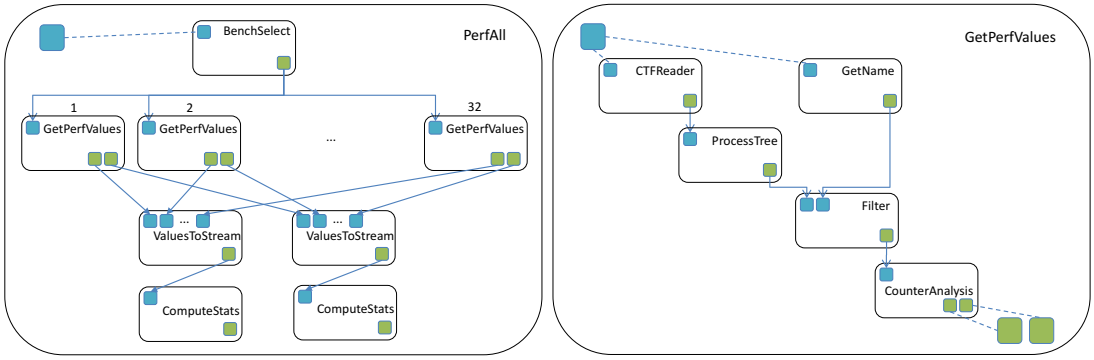
4.1. The Phoronix Test Suite

The *Phoronix Test Suite* (*PTS*) [15] provides a set of benchmarks targeting different aspects of a system. *PTS* is available on multiple platforms including Linux, MacOS, Windows, Solaris and BSD.

PTS comes with some 200 open-source test programs. It includes hardware benchmarks typically testing battery consumption, disk performance, processor efficiency or memory consumption. It also targets diverse environments including OpenGL, Apache, compilers, games and many others. Each benchmark produces a *score* which is the value of the metric it investigates. The `compress-gzip`benchmark, for example, characterizes processor performance by measuring the time, in seconds, needed for compressing a file. Another example is `iozone`which provides a Mb/s score for disk I/O performance.

PTS provides little information about benchmarks' logic and internals. Each benchmark is tagged as one of *Disk*, *Graphics*, *Memory*, *Network*, *Processor* and *System*, supposedly to indicate which system part is tested, but there is no further information on how this tag has been decided or how exactly the benchmark tests this system part.

DOI: 10.1002/spe



(a) (b)

```
1 class PerfAll: virtual public Module {
2 ...
3   void compute(){
4
5     int num_runs = number_of_runs;
6
7     Workflow w;
8
9     ValuesToStream<uint64_t> *vts_mem_instr = new ValuesToStream<uint64_t> (num_runs);
10    ValuesToStream<uint64_t> *vts_cpu_instr = new ValuesToStream<uint64_t> (num_runs);
11    w.addModule(*vts_mem_instr);
12    w.addModule(*vts_cpu_instr);
13
14    for(int i=0; i<num_runs; i++){
15      GetPerfValues *gpv = new GetPerfValues;
16      w.addModule(*gpv);
17      w.link(gpv->mem_instr, vts_mem_instr->data[i]);
18      w.link(gpv->cpu_instr, vts_cpu_instr->data[i]);
19    }
20
21    // mem
22    ComputeStats<int64_t> *mem_instr_stats = new ComputeStats<int64_t>;
23    w.addModule(*mem_instr_stats);
24    w.link(vts_mem_instr->datastream, mem_instr_stats->data_in);
25    // cpu
26    ComputeStats<int64_t> *cpu_instr_stats = new ComputeStats<int64_t>;
27    w.addModule(*cpu_instr_stats);
28    w.link(vts_cpu_instr->datastream, cpu_instr_stats->data_in);
29    ...
30    w.start();
31 }};
```

(c)

Figure 6. Zooming into the `PerfAll` module

(a) the nested workflow in the module `PerfAll` using `GetPerfValues` modules
(b) the nested workflow in the `GetPerfValues` module
(c) the code of the `PerfAll` module

The repartition of the benchmarks is highly irregular. If we consider that *PTS* benchmarks having the same tag form a benchmark family, the *Network* family contains only one test, while the *Processor* family contains 90 tests (Table II). If, in the first case, a developer has no choice, in the second case, he/she will need to know more about the benchmarks to choose the most relevant.

Family	*System*	*Processor*	*Network*	*Memory*	*Graphics*	*Disk*
Number of benchmarks	31	90	1	2	75	13

Table II. Repartition of the *PTS* benchmarks

DOI: 10.1002/spe

4.2. Experimental Setup

We have worked with 10 *PTS* bechmarks representing the different *PTS* families. We have chosen them among the most popular *PTS* benchmarks as determined by the number of downloads and available results [58]. Table III gives a brief description for each benchmark and specifies the corresponding score metric.

Benchmark	Version	PTS family	Description	Score metric
`compress-gzip`	1.1.0	*Processor*	File compression	`s`
`ffmpeg`	2.5.0	*Processor*	Audio/video encoding	`s`
`scimark2`	1.2.0	*Processor*	Scientific computations	`Mflops`
`stream`	1.3.0	*Memory*	Memory I/O performance	`MB/s`
`ramspeed`	1.4.0	*Memory*	Memory I/O performance	`MB/s`
`phpbench`	1.1.0	*System*	PHP interpreter benchmarking	`a number` (higher is better)
`pybench`	1.0.0	*System*	Python performance	`ms`
`iozone`	1.8.0	*Disk*	Disk I/O performance	`MB/s`
`unpack-linux`	1.0.0	*Disk*	Unpack the Linux kernel	`s`
`network-loopback`	1.0.1	*Network*	Network loopback performance	`s`

Table III. Benchmark list

Each benchmark is run with its default options as defined by the *PTS* system except for the number of runs (32 instead of 3). The experiments have been run on four different platforms which helped validate the fact that benchmarks have similar executions, hence profiles, whatever the platform. We have used one UDOO board [59], one Juno board [60], one nvidia Jetson TK1 board [61] and a desktop machine. In the following sections we show results from the Juno board and the desktop machine.

The Juno board has one dual core Cortex-A57 processor, one quad core Cortex-A53 processor and 8GB of memory. The desktop machine has a x86-64 Xeon E3-1225@3.20GHz processor and 32GB of memory. Both platforms have a Gigabit Ethernet connection and use SSD storage. In terms of software, the two platforms use the same Debian version with the `4.3.0-1-arm64` kernel for the Juno board and the `4.4.0-1-amd64` kernel for the desktop machine. On both we have used the stable version 2.8 of LTTng, version 6.2.2 of Phoronix and version 5.3.1 of `gcc`.

4.3. Tracing Overhead and Benchmark Stability

To verify whether the captured LTTng traces are representative of the benchmarks' behavior, we have investigated LTTng overhead. A negligible overhead is a prerequisite to producing traces that reflect normal benchmark execution and that are worth analyzing. We have run the benchmarks with and without tracing and have compared the respective benchmarks's scores, as well as execution times.

We have used five different execution configurations. The `baseline` configuration corresponds to running the benchmark without tracing. The `time` configuration measures the benchmark's execution time with the `/usr/bin/time` utility. To minimize LTTng overhead, instead of tracing all needed data during a single execution, we have considered three different tracing configurations. `all-events`, is used to trace the kernel. `libc` is used to trace memory-related function calls. Finally, `perf`, provides the hardware counter information.

The score variations of the `ramspeed`, `compress-gzip`, `pybench` and `network-loopback` benchmarks are given in Table IV. We omit the results for the other six benchmarks as they are quite similar. For each execution configuration, we provide the mean score, the standard deviation and compare this mean score to the reference score obtained by the `baseline` configuration. For example, the `all-events` configuration of the `ramspeed` benchmark produces a mean score of $14565,5$ `MB/s` with a standard deviation of $0,08$. The difference between this mean score and the one obtained by the `baseline` configuration represents $0,49\%$ of the latter which means that the scores are quasi-equal.

DOI: 10.1002/spe

Benchmark (score metric)	Configuration	Mean	Std. dev	Tracing impact (%)
ramspeed (MB/s)	baseline	14 637.1	0.04	
	all-events	14 565.5	0.08	0.49
	libc	14 636.7	0.06	0.00
	perf	14 628.9	0.03	0.06
	time	14 632.3	0.05	0.03
compress-gzip (s)	baseline	11.28	0.54	
	all-events	11.47	0.37	1.61
	libc	11.51	0.09	2.00
	perf	11.30	0.63	0.21
	time	11.25	0.65	0.29
pybench (ms)	baseline	1627.88	0.27	
	all-events	1631.47	0.34	0.22
	libc	2284.75	0.38	40.35
	perf	1628.88	0.29	0.06
	time	1629.59	0.43	0.11
network-loopback (s)	baseline	9.39	0.68	
	all-events	15.51	6.45	65.05
	libc	15.28	4.06	62.62
	perf	18.91	4.82	101.27
	time	9.37	0.65	0.29

Table IV. Benchmarks' scores variations (x86)

If we focus on the last column of Table IV, we observe that, in most cases, the tracing impact is less that 2%. This means that tracing configurations do not perturb the benchmark execution and yield the same score. Indeed, if we look at the `compress-gzip` benchmark, for example, the scores obtained for the `all-events`, `libc`, `perf` and `time` configurations are all very close to the score of $11,28$s obtained during a standard run.

We do observe score perturbations for the `libc` configuration of the `pybench` benchmark and the tracing configurations of `network-loopback`. In the first case, the `pybench`ś score becomes $2284,75$ms which is 40% slower than the `baseline`'s $1631,47$ms. This perturbation can be explained by the heavy usage of memory-related functions and the cost of the function interception mechanism. In the case of `network-loopback`, tracing has an important impact as it may double the score (`perf` configuration). This is related to the huge number of traced events, around 4.10^8 per run.

To consider tracing perturbations over the benchmarks' execution times, we have compared the three tracing configurations to the `time` configuration. Table V gives a sample from the obtained results on the desktop machine, the results on the Juno board being quite similar. The major observation is that the tracing of memory-related functions through interception (`libc` configurations) is costly in all cases and may really slow down the benchmarks' execution. For the benchmarks shown in Table V, the execution is up to three times slower.

Concerning the other tracing configurations, most of the considered benchmarks exhibit similar results to `pybench` : the tracing of both hardware counters and kernel events do not slow down the execution. In the case of the `compress-gzip` benchmark, the observed slow down may be explained by the fact that this benchark has an important kernel activity, as discussed in the next section. Finally, the only benchmark that presents considerable slow down and duration variations is `network-loopback`which is again explained by the important number of traced events.

The conclusions we can draw from the investigation of LTTng overhead and the obtained results are that we can safely use execution traces to characterize benchmarks, even if tracing may be costly. Indeed, the presented results show that tracing does not impact benchmarks' scores and therefore does not perturb their execution. Moreover, mean values are representative, as benchmarks' execution is deterministic. This is attested by the negligible variations of their scores (standard deviation in Table IV), durations (standard deviation in Table V), as well as all other profile metrics (standard deviation of less that 1%). For example, if we consider the number

DOI: 10.1002/spe

Benchmark	Configuration	Mean duration (s)	Std. dev	Tracing impact (%)
pybench	time	36.89	0.52	
	all-events	36.99	0.50	0.28
	libc	65.15	0.32	76.62
	perf	36.88	0.30	0.02
compress-gzip	time	13.46	1.50	
	all-events	19.07	2.11	41.64
	libc	28.91	8.67	114.77
	perf	14.54	0.85	8.00
network-loopback	time	10.03	1.82	
	all-events	16.27	6.28	62.19
	libc	29.25	10.46	191.67
	perf	19.03	17.97	89.78

Table V. Benchmarks' duration variations (x86)

of kernel events for the compress-gzip benchmark, we obtain Table VI. If we consider the first line, compress-gzip execution contains around 4130 disk-related events (around $0,18\%$ of all traced events) and the 32 executions differ by 1 or 2 events! Finally, the analysis about benchmarks' durations estimates the time cost of trace production. If tracing the kernel activity of pybenchdoes not slow down its execution, this is not the case for compress-gzip or network-loopback. A benchmark analyst may choose to *not* profile a certain aspect of a target benchmark if the respective execution traces are too costly to produce.

Event type	Mean number of events of this type	% of the total number of events	Standard deviation (number of events)
Disk	4130	0.18	1.73
Memory	1 842 820	80.74	0.5
Network	133	0.01	1.46
Processor	17 489	0.77	3.7
System	417 915	18.31	1.4

Table VI. Kernel operation of compress-gzip (x86)

4.4. *Kernel vs User Time*

A first simple classification of Phoronix benchmarks is to consider the ratio of kernel versus user operation. Figure 7 reflects this ratio for the benchmark executions on the desktop machine, the Juno results being the same. As explained earlier, the ratio here is computed over benchmarks' useful execution time and ignoring the idle time.

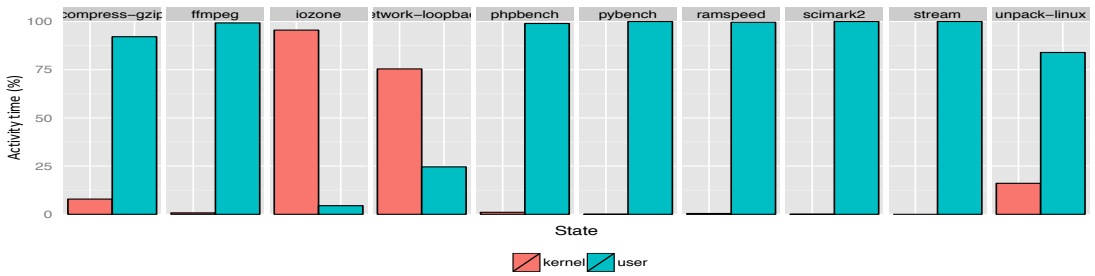


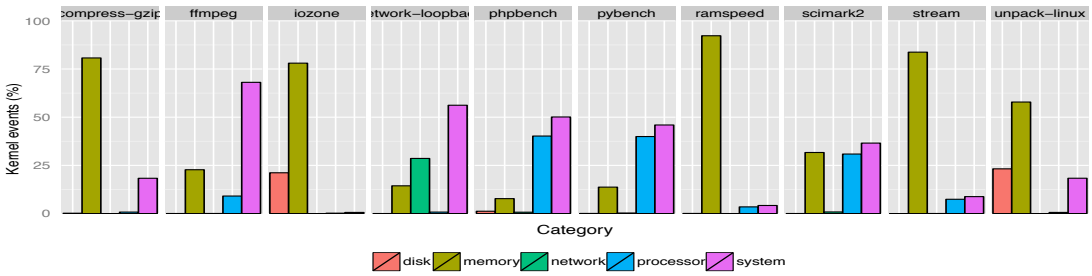
Figure 7. User vs kernel time (x86 desktop machine)

We can see that only the two benchmarks that execute multiple I/O operations, namely iozone and network-loopback, spend most of the time in kernel mode. The other eight benchmarks execute predominantly in user mode.

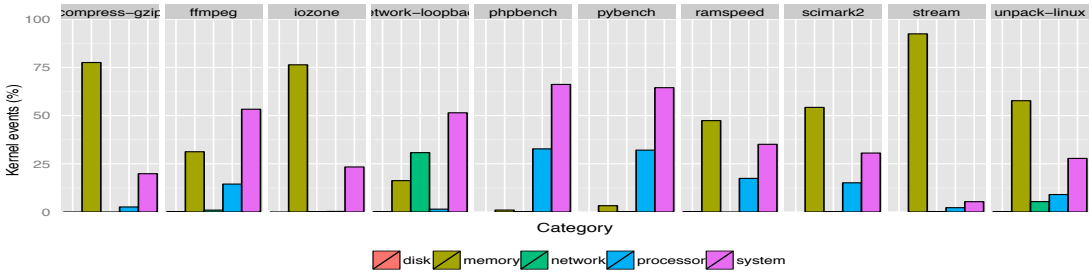
DOI: 10.1002/spe

4.5. Benchmark Type

4.5.1. Kernel operation. To gain more insight into benchmarks' kernel operation, we consider the repartition of kernel events (*Processor*, *Memory*, *Disk*, *Network* and *Graphics*) and investigate which part of the kernel is mostly used. This is especially useful for understanding the behavior of benchmarks spending most of their time in kernel mode. The kernel profiles are shown in Figure 8.



(a) x86 desktop machine



(b) Juno board

Figure 8. Kernel operation

Looking at the results concerning the desktop machine (Figure 8a), we can clearly see that `network-loopback` tests the network as it is the only benchmark exhibiting network events. Similarly, `iozone` and `unpack-linux` prove to test, as announced by *PTS*, the disk, as they are the two benchmarks to have disk events.

If we consider the benchmarks tagged as *Memory* within *PTS*, namely `stream` and `ramspeed`, we find two very similar kernel profiles confirming the important usage of memory-related functions. To choose between the two benchmarks, it may be interesting to consider the one with shorter execution time.

In the *Processor* family, `ffmpeg`, `scimark2` and `compress-gzip` have very different profiles. `compress-gzip` has a predominant number of memory events, `ffmpeg` counts many system events and only `scimark2` shows an important number of processor events. This is clearly an example where the *PTS* tag is not informative enough and where an analyst would need more information on benchmark operation in order to make his/her choice.

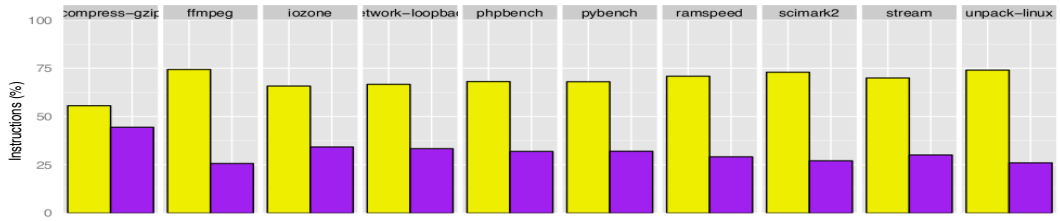
Our analysis of the *System* Phoronix family made us understand that it includes various benchmarks testing different software systems (layers, middleware) and does not necessarily focus on the operating system level. In the considered set of benchmarks, `phpbench` and `pybench` have similar kernel profiles and do exhibit an important number of system events. However, they respectively test the performances of PHP and Python code.

For most benchmarks, the kernel profiles obtained on the two execution platforms do not differ. In the case of the *Disk* benchmarks, however, we do not observe any disk events on the Juno board which comes from its management and optimizations of I/O operations.

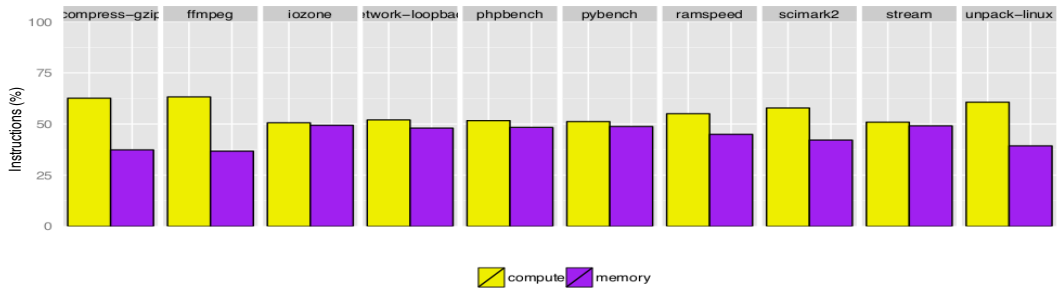
4.5.2. User operation. User-level operation is reflected in the results obtained about the values of the instruction and cache-related hardware counters (Figure 9). If we consider the desktop machine

DOI: 10.1002/spe

(Figure 9a), we see that the number of computational and memory instructions are difficult to correlate to the *PTS* families. In the cases of the `ffmpeg` and `scimark2` benchmarks, part of the *Processor* family, we do observe more computations than memory accesses (around 75% against 25%). However, this is not true for the `compress-gzip` benchmark which executes multiple memory operations. `compress-gzip` has about 46% of memory-related operations which is actually more than what we find for the `stream` and `ramspeed` *Memory* benchmarks. Globally, we observe that the number of computation instructions is two to three times bigger than the number of memory-related instructions. This can be explained by the fact that on the x86 machine data accesses can be done during computation instructions. This is not the case on the Juno platform (Figure 9b) where computations and memory accesses are more balanced.



(a) x86



(b) Juno board

Figure 9. Ratio between computation and memory instructions

4.6. Memory Usage Profile

If hardware counters do not yield much information about the memory behavior of a benchmark, the interception of the memory-related function calls provides a much needed insight. The results concerning the maximum allocated memory and the number of null `free` calls are given in Table VII. The evolution of the benchmarks' used memory is given in Figure 10.

The first conclusion we can draw is that the results are stable across platforms. In both the table and the figure, if we compare the Juno results (left) to the x86 results (right), the memory metrics do not really vary.

A second observation from Figure 10 is that we witness several quite different memory profiles. `scimark2` operates in phases during wich the amount of used memory is stable. However, the duration and the used memory are quite different across the different phases. After an initial short burst in memory allocation, the memory usage of `ffmpeg` is quite stable during the entire benchmark execution. In the case of `stream` the execution is punctuated by regular, maybe even periodic, memory allocations and liberations. `ramspeed` steadily increases its memory consumption, while, finally, `unpack-linux` allocates the needed memory in the beginning of its execution to free it at the end.

The calls to `free(null)` are not errors but may indicate code optimization or performance issues. If in most cases the number of such calls is negligible, it may be useful to investigate the reasons of their presence (more than 200000 calls) in the `ffmpeg` benchmark.

DOI: 10.1002/spe

If we consider the maximum memory allocated by benchmarks (second and third columns in Table VII), we see that the benchmarks from the same *PTS* family have quite different behavior. In the *Processor* family, `compress-gzip` uses a small amount of memory (138 KB), `scimark2` uses quite a lot of memory (16 GB) and `ffmpeg` is in the middle (5 GB). If `compress-gzip` and `ffmpeg` show a stable memory usage (cf. `ffmpeg`'s profile in Figure 10c), `scimark2` exhibits distinctive execution phases (Figure 10a).

In the case of the considered *Memory* benchmarks, `ramspeed` allocates up to 3.5 GB of memory while `stream` allocates only 10 KB. The memory evolution profiles in Figure 10 show that if both benchmarks do stress the memory, `ramspeed` tests memory allocation, while `stream` tests heavy memory accesses to a localized memory region.

Benchmark	Maximum allocated memory (KB)		# `free(null)`	
	Juno	x86	Juno	x86
`compress-gzip`	138	134	4	5
`ffmpeg`	5 215	5 215	216 544	215 270
`iozone`	37 751	37 753	3	3
`network-loopback`	1 063	1 033	317	95
`phpbench`	4 572	4 577	45	45
`pybench`	2 863	2 864	158	42
`ramspeed`	3 456 110	3 456 110	0	4
`scimark2`	16 780	16 780	0	0
`stream`	10	10	44	404
`unpack-linux`	3 697	3 705	495 306	494 910

Table VII. Memory usage

4.7. CPU Usage and Parallelization

The way benchmarks use available processors on the desktop machine is shown in Figure 11. If we look at the mean values (Figure 11a), we see that the `stream` benchmark occupies the four processors at almost 100%. The `ffmpeg` benchmark also balances the load equally among the four processors and uses them 70% of the time. For the other benchmarks, the processors are used at around 50% which means that the rest of the time they are idle. In the particular case of `iozone` which tests the *Disk*'s performance, the CPU usage is even lower and each CPU is used at most 25% of the time.

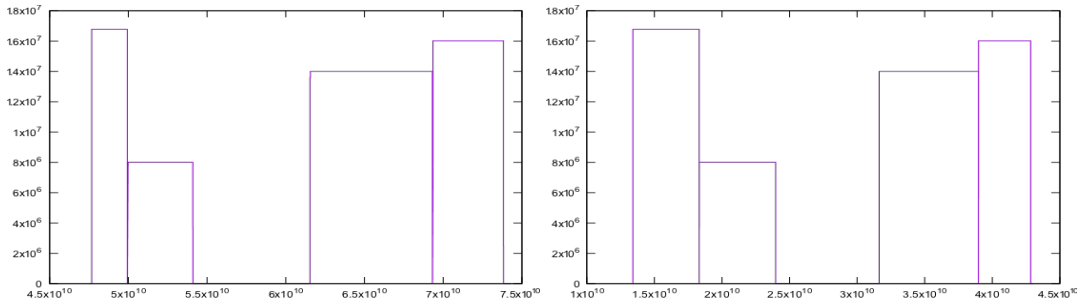
The mean CPU usage values do not show how exactly the four processors of the desktop machine are used. If we consider a single benchmark execution (Figure 11b), we obtain a different picture for most of the benchmarks. If we focus on the `compress-gzip` benchmark, we find that it is sequential as it uses almost exclusively a single processor. The fact that over the 32 runs the four processors are equally used shows that the scheduling is not deterministic and that the processor to execute the benchmark changes from one run to another. The same reasoning applies to `phpbench` `pybench` and `scimark2`. `ramspeed` and `unpack-linux` prove to use only two out of the four processors. Finally, the benchmarks that really exploit the platform parallelism reveal to be `ffmpeg` `network-loopback` and `stream`.

If we compare these results with an execution on the Juno board (Figure 11c), we see that the parallel benchmarks (`ffmpeg`, `network-loopback` and `stream`) continue to use all the available processors. However, the differences between the Juno's CPUs (0 and 1 are $A57$, 2 to 5 are $A53$) result in an unbalanced CPU usage. `ramspeed` which uses only two processors on the desktop machine exhibits a more parallel execution on the Juno board. As for the `compress-gzip`, `phpbench` and `pybench` sequential benchmarks, the CPU which is scheduled is one of the less performant ($A53$). The corresponding benchmarks' scores do not therefore take into account platform heterogeneity and do not report the best possible results.

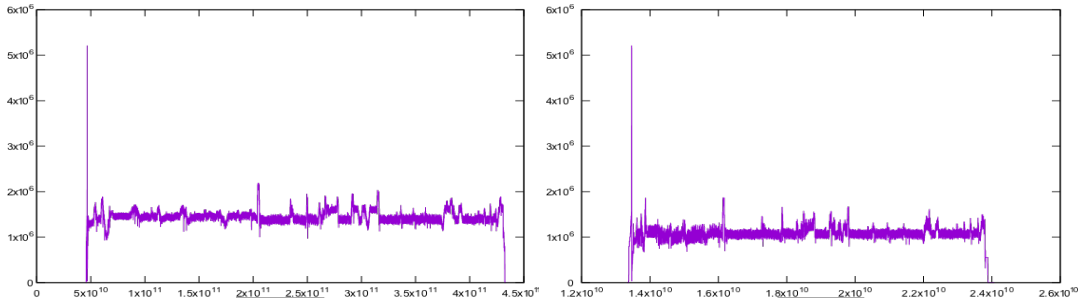
4.8. SWAT Performance

Table VIII resumes the data concerning the produced execution traces and their analysis on the desktop machine. For each benchmark we give the size of the traces, the number of traced events,

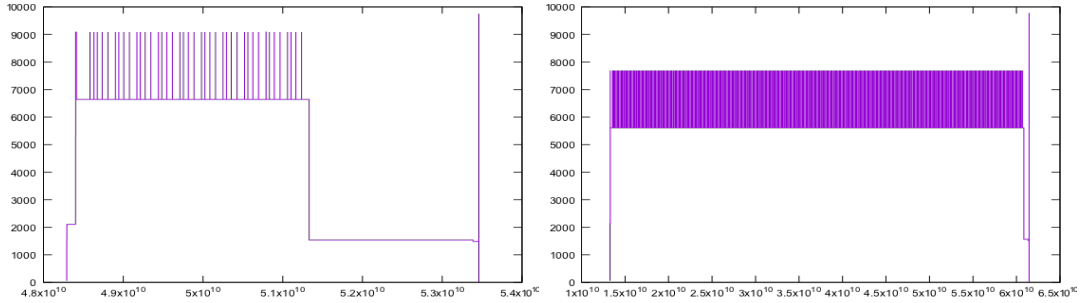
DOI: 10.1002/spe



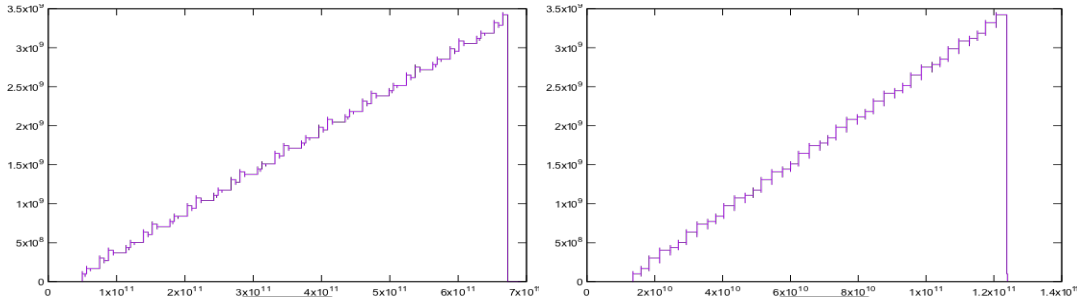
(a) `scimark2` (Juno) (b) `scimark2` (x86)



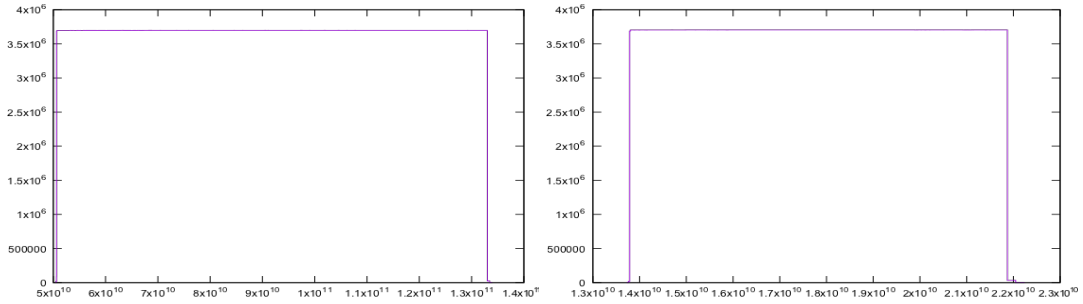
(c) `ffmpeg` (Juno) (d) `ffmpeg` (x86)



(e) `stream` (Juno) (f) `stream` (x86)



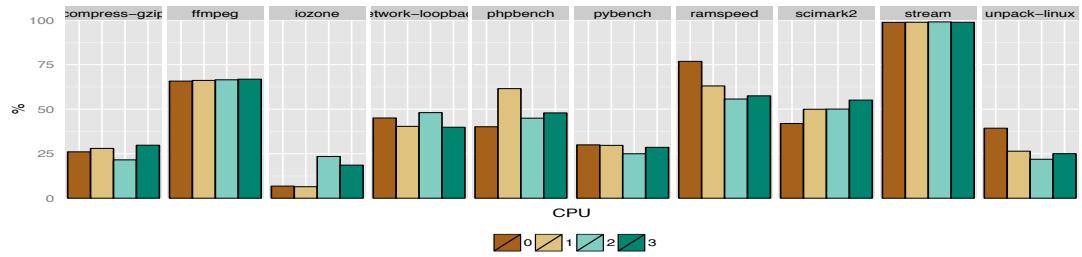
(g) `ramspeed` (Juno) (h) `ramspeed` (x86)



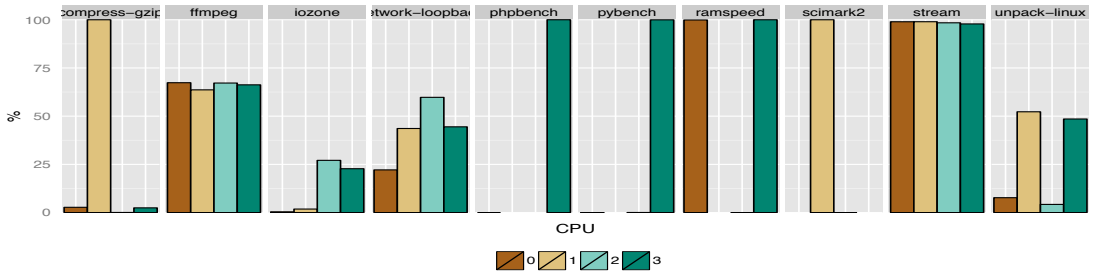
(i) `unpack-linux` (Juno) (j) `unpack-linux` (x86)

Figure 10. Memory evolution profiles (Bytes versus Time (ns))

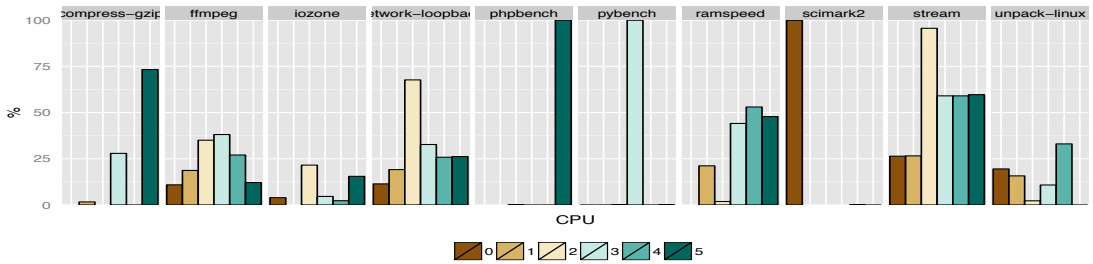
DOI: 10.1002/spe



(a) Mean CPU usage over 32 runs (x86)



(b) CPU usage for one run (x86)



(c) CPU usage for one run (Juno)

Figure 11. CPU usage

the time needed to analyze the traces and the time spent to execute the benchmarks and produce the traces. The given values sum these metrics for the 32 runs of the benchmarks (3 tracing configurations multiplied per 32 runs). If we consider the `compress-gzip` benchmark, for example, generating the traces has taken around 40 minutes (2431 seconds) and their analysis has lasted for about 2 hours (6626 seconds)

If we sum the values from the first and the second columns, we obtain respectively that we have generated about $500GB$ of tracing data containing 25.10^9 events. As supposed, the time needed for trace analysis is not a function of the execution time but depends on the number of traced events. In the case of `compress-gzip`, `iozone` or `pybench`, for example, the analysis is two times slower than the execution. In the case of the `network-loopback` benchmark, the analysis is expensive because of the important number of trace events. On the contrary, the analysis of `scimark2` 's traces is fast as there are less trace events. Globally, our analysis workflow has been able of processing around 200000 events per second. Compared to our first workflow implementation which uses the standard approach of storing traces in a database, the SWAT prototype is about 5 times faster.

DOI: 10.1002/spe

Benchmark	Traced data (MB)	# events	Analysis (s)	Execution (s)
compress-gzip	43 166	1899×10^6	6 626	2 431
ffmpeg	4 720	221×10^6	1 018	1 927
iozone	17 227	770×10^6	3 231	2 494
network-loopback	266 652	13346×10^6	69 369	2 386
phpbench	47 788	2790×10^6	14 595	9 927
pybench	45 222	2732×10^6	12 347	5 629
ramspeed	32 094	1540×10^6	6 074	29 640
scimark2	1 921	111×10^6	352	4 272
stream	6 289	330×10^6	1 580	5 162
unpack-linux	14 139	643×10^6	2 930	1 683
Total	479 218	24382×10^6	118 122	65 551

Table VIII. Tracing information (x86)

The traces produced for our analysis have been made publicly available at zenodo.org [§¶‖] and at https://persyval-platform.imag.fr [**]. The SWAT framework is available at gitlab.com[††].

4.9. *Lessons learned with SWAT*

In this section we provide a synthetic view on the usefulness of the SWAT workflow and the way it can be used to reason about benchmarks. We reference results presented in the previous sections, as well as results that have not been explicitly listed.

The investigations on tracing overhead in Section 4.3 have shown that SWAT can be used to reason about benchmarks as the production of their traces does not perturb their normal behavior. However, the cost of tracing and of trace analysis, as discussed in Section 4.8, is not to be neglected. When choosing a benchmark, a system analyst should either know in advance the expected behavior of a benchmark, or be willing to spend the necessary storage and time resources to investigate it. In the case of the phpbench benchmark, for example, SWAT has required $9\,927s$ for tracing and $14\,595s$ for trace analysis which resulted in about $7h$ of benchmark profiling!

The *stability* issues discussed in Section 4.3 are important as benchmarks are expected to be deterministic and reproduce the same results over different runs. Benchmarks exhibiting important variations cannot be used to characterize a target system. They should either be ignored or be further investigated. Indeed, score, duration or metric variations may point out singular yet important phenomena in the target system behavior. In our experiments, nine out of ten benchmarks have presented stable scores as their standard deviation is close to 0. In the case of the iozone benchmark, the mean disk I/O performance on the x86 platform has been measured to be $354,83$MB/s. However, this is closer to the maximum obtained value ($365,34$MB/s) than to the minimum ($268,76$MB/s) one. We can conclude that few executions are less performant because of some system perturbation. This perturbation is to be identified if the benchmark is to be reliably used.

Not knowing the internal behavior of a benchmark hinders the interpretation of the benchmark result. For example, if we use both ramspeed and stream benchmarks on the x86 platform, the first measures memory I/O performance at around 14GB/s (Table IV), while the second yields around $16,8$GB/s. To understand the difference, one would need the memory profiles provided by SWAT. Not knowing the difference and testing a system with ramspeed rather than with

[§] https://doi.org/10.5281/zenodo.437170
[¶] https://doi.org/10.5281/zenodo.437179
[‖] https://doi.org/10.5281/zenodo.437207
[**] https://doi.org/10.18709/PERSCIDO.2017.03.DS31
[††] https://gitlab.com/alexmartin/swat

DOI: 10.1002/spe

`stream` will result in worse performance results. In the case of processor performance, ignoring the SWAT CPU usage profiles (Figure 11) may even lead to wrong results. An analyst interested in the performances of the x86 processor of the Juno board may actually get a measure for the ARM processor if he/she uses the `scimark2`. As this benchmark is sequential (Figure 11b and Figure 11c), its result will vary according to the processor on which it is scheduled. To conclude, if the optimal processor performance is to be characterized, an analyst should use a benchmark that is capable of taking advantage of the computational resources through a good parallelization or a selection of the most efficient CPUs.

The information on CPU usage and memory consumption may also be used as quantitative metrics for benchmarks. An analyst may opt for "light" benchmarking that demands a limited amount of resources or, on the contrary, target heavy use by allocating the maximum resources.

The duration metric is a simple indicator for the time cost of system benchmarking. Among benchmarks targeting the same system aspect and providing the same metric, it is natural to prefer the shorter ones.

Finally, the information on the user versus kernel time (Section 4.4) and the kernel operations (Section 4.5) shed light on the benchmarks' capacity to isolate the aspects they test. The SWAT profiles show which system parts are involved during benchmarking and thus reveal possible side effects or perturbations.

5. CONCLUSION AND PERSPECTIVES

We have presented in this paper a workflow-based tool for automatic profiling of benchmarks. The result is a unified profile which characterizes a benchmark using intuitive metrics including duration, CPU utilization, parallelization, stability and memory usage. These metrics allow for simple comparison among benchmarks targeting the same type of systems and thus facilitate the choices of a system performance analyst. We have illustrated our approach with the Phoronix Test Suite and have experimented with embedded and desktop Linux-based platforms. We have successfully produced the profiles for multiple benchmarks exhibiting their different characteristics. The automatic nature of our profiling makes it applicable *as is* in the context of other benchmarks. The fact that the SWAT prototype works with LTTng-obtained CTF traces opens the way to sharing our experience with a greater community by integrating our trace analysis in the mainstream developments.

In our work we have taken advantage of workflows' useful features such as automatisation, result caching and reproducibility. However, most workflow systems do not properly address the data management issues when it comes to manipulating big data sets. In this regard, we have shown that workflow tools should provide for pipelining, streaming and parallel computations. An ongoing collaboration with the VisTrails team investigates the way these features may be brought to the VisTrails tool [62, 45]. An interesting perspective is to evaluate the applicability and the cost of using *big data* streaming tools such as Apache Storm or Spark [63, 64].

As the execution time of the SWAT workflow is proportional to the size of execution traces, its performance may be optimized with efficient event filtering. Instead of collecting and using full kernel traces, it would be possible to consider only the events needed for the computation of the target performance metrics [49, 65]. Trace reduction, however, will translate into the oblivion of execution events and therefore make some analyses impossible.

The profiles computed by our solution are benchmark-agnostic and therefore do not reflect benchmarks' specifics. A long term research objective would be to provide generic means for taking into account the benchmarking context and identify benchmarks' domain-specific distinguishing features.

Our experimentation has put forward the fact that the initially provided description is not sufficient to understand the way benchmarks test the target system. Even if our profiling solution does succeed in obtaining useful insight about a benchmark, it comes with a cost in terms of time, computation and storage and cannot supplant a clear specification provided by the bechmark's designer. We believe that an explicit effort is to be made to provide open benchmarks with clear function

DOI: 10.1002/spe

descriptions. This is especially true in the cases where the final system to design, be it a software or a hardware one, is to provide performance guarantees in specific application domains. In the domain of high performance computing (HPC), for example, the scientists working on weather, nuclear or astrophysical simulations invest in supercomputers that are to specifically respond to their application needs. To dimension the hardware and to design its software, it is primordial to have benchmarks that are representative of the target computations and workloads. Only with clear benchmark descriptions the benchmarking effort may be relevant and reproducible and prevent the bias of Goodhart's law which states that "When a measure becomes a target, it ceases to be a good measure.".

REFERENCES

1. Almeida J, Frade M, Pinto J, Melo de Sousa S. *Rigorous Software Development: An Introduction to Program Verification*. Undergraduate Topics in Computer Science, Springer-Verlag London, 2011; 15–44.
2. Intel. Intel MPI Benchmarks 4.0 Update 2. URL https://software.intel.com/en-us/articles/intel-mpi-benchmarks.
3. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. URL http://mvapich.cse.ohio-state.edu/benchmarks/.
4. Future Benchmarks and Performance Tests. URL http://www.futuremark.com/.
5. ApacheBench. URL http://httpd.apache.org/docs/2.2/programs/ab.html.
6. HP. The httperf Tool. URL http://www.hpl.hp.com/research/linux/httperf/.
7. Linux Benchmark Suite Homepage. URL http://lbs.sourceforge.net/.
8. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press: New York, NY, USA, 2000.
9. Dongarra J, Martin J, Eigenmann R. Benchmarks. *Encyclopedia of Computer Science*. John Wiley and Sons Ltd.: Chichester, UK; 137–141. URL http://dl.acm.org/citation.cfm?id=1074100.1074163.
10. Huppler K. *The Art of Building a Good Benchmark*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2009; 18–30, doi:10.1007/978-3-642-10424-4_3.
11. Vieira M, Madeira H, Sachs K, Kounev S. *Resilience Benchmarking*. Springer Berlin Heidelberg: Berlin, Heidelberg, 2012; 283–301, doi:10.1007/978-3-642-29032-9_14.
12. Kistowski J, Arnold JA, Huppler K, Lange KD, Henning JL, Cao P. How to build a benchmark. *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, ACM: New York, NY, USA, 2015; 333–336, doi:10.1145/2668930.2688819.
13. Martin A. An Infrastructure for the generic and optimized analysis of execution traces of embedded systems. Theses, Université Grenoble Alpes Jan 2017. URL https://hal.archives-ouvertes.fr/tel-01492474.
14. Martin A. SWAT : Système de Workflow pour l'Analyse de Traces d'exécution. URL https://gitlab.com/alexmartin/swat.
15. Phoronix Test Suite. URL http://www.phoronix-test-suite.com/.
16. DBench. URL https://www.samba.org/ftp/unpacked/dbench/.
17. The Linpack Benchmark. URL https://www.top500.org/project/linpack/.
18. TPC Benchmarks. URL http://www.tpc.org.
19. SPEC. URL https://www.spec.org.
20. Dependability Benchmarking Project. URL http://webhost.laas.fr/TSF/DBench/.
21. NAS Parallel Benchmarks. URL https://www.nas.nasa.gov/publications/npb.html.
22. EPCC OpenMP micro-benchmark suite. URL https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking.
23. Industry-Standard Benchmarks for Embedded Systems. URL http://www.eembc.org/benchmark/products.php/.
24. Joshi A, Member S, Phansalkar A, Member S. Measuring Benchmark Similarity Using Inherent Program Characteristics. *IEEE Transactions on Computers* 2006; **55**(6):769–782.
25. gprof. URL https://sourceware.org/binutils/docs/gprof/.
26. OProfile. URL http://oprofile.sourceforge.net/.
27. Performance Application Programming Interface. URL http://icl.utk.edu/papi/.
28. Seward J, Nethercote N, Weidendorfer J. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008.
29. Janjusic T, Kartsaklis C. Glprof: A gprof inspired, callgraph-oriented per-object disseminating memory access multi-cache profiler. *Procedia Computer Science* 2015; **51**:1363 – 1372, doi:http://dx.doi.org/10.1016/j.procs.2015.05.324. International Conference On Computational Science, {ICCS} 2015Computational Science at the Gates of Nature.
30. Lachaize R, Lepers B, Quema V. Memprof: A memory profiler for NUMA multicore systems. *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, USENIX: Boston, MA, 2012; 53–64. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/lachaize.
31. Prada-Rojas C, Santana M, De-Paoli S, Raynaud X. Summarizing Embedded Execution Traces through a Compact View. *Conference on System Software, SoC and Silicon Debug S4D*, 2010.

DOI: 10.1002/spe

32. Marangozova-Martin V. Towards Generic System Observation Management Jun 2015. URL https://hal.inria.fr/tel-01171642.
33. STMicroelectronics. STWORKBENCH. URL http://www.st.com/en/development-tools/stworkbench.html.
34. STMicroelectronics. STLinux Trace Viewer Trace Format. URL http://www.stlinux.com/stworkbench/interactive_analysis/stlinux.trace/kptrace_traceFormat.html.
35. Trace Compass. URL http://tracecompass.org/.
36. CTF : Common Trace Format. URL http://www.efficios.com/ctf.
37. Scalasca. URL http://www.scalasca.org/.
38. Knüpfer A, Brunst H, Brendel R. Open Trace Format. *Specification*, Center for Information Services and High Performance Computing (ZIH) Technische Universität Dresden, Germany Nov 2011.
39. Linux Trace Toolkit: next generation. URL http://lttng.org.
40. Giraldeau F, Desfossez J, Goulet D, Desnoyers M, Dagenais MR. Recovering System Metrics from Kernel Trace. *Linux Symposium 2011*, 2011.
41. Cohen-Boulakia S, Leser U. Search, adapt and reuse: The future of scientific workflows. *SIGMOD Records* 2011; **40**(2):1187–1189.
42. Qin J, Fahringer T. *Scientific Workflows, Programming, Optimization, and Synthesis with ASKALON and AWDL*, vol. ISBN 978-3-642-30714-0. Springer, 2012.
43. Kim J, Gil Y, Spraragen M. A knowledge-based approach to interactive workflow composition. *IN PROCEEDINGS OF THE 2004 WORKSHOP ON PLANNING AND SCHEDULING FOR WEB AND GRID SERVICES, AT THE 14TH INTERNATIONAL CONFERENCE ON AUTOMATIC PLANNING AND SCHEDULING (ICAPS 04)*, 2004.
44. Deelman E, Vahi K, Juve G, Rynge M, Callaghan S, Maechling PJ, Mayani R, Chen W, da Silva RF, Livny M, *et al.*. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 2015; **46**:17 – 35, doi:https://doi.org/10.1016/j.future.2014.10.008.
45. VisTrails. URL www.vistrails.org.
46. Desnoyers M. Low-Impact Operating System Tracing. PhD Thesis 2009.
47. Chen KY, Chang YH, Liao PS, Yew PC, Cheng SW, Chen TF. Selective Profiling for OS Scalability Study on Multicore Systems. *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications* 2013; :174–181doi:10.1109/SOCA.2013.28.
48. lttng-calibrate. URL http://lttng.org/man/1/lttng-calibrate/v2.8/.
49. Sharma SD, Dagenais M. Enhanced userspace and in-kernel trace filtering for production systems. *J. Comput. Sci. Technol.* 2016; **31**(6):1161–1178, doi:10.1007/s11390-016-1690-y.
50. Linux Perf Tool. URL https://perf.wiki.kernel.org/.
51. Chen H, Wagner D, Dean D. Setuid demystified. *Proceedings of the 11th USENIX Security Symposium*, USENIX Association: Berkeley, CA, USA, 2002; 171–190. URL http://dl.acm.org/citation.cfm?id=647253.720278.
52. Jain R. *The Art Of Computer Systems Performance Analysis*. 1991.
53. The SoC-TRACE Project. URL http://soc-trace.minalogic.net.
54. STMicroelectronics. STLinux. URL http://www.st.com/en/development-tools/stlinux.html.
55. gstreamer: Open Source Media Framework. URL https://gstreamer.freedesktop.org/.
56. Wang AJA, Qian K. *Component-Oriented Programming*. Wiley-Interscience, 2005.
57. Babeltrace. URL http://man7.org/linux/man-pages/man1/babeltrace.1.html.
58. Phoronix Test Suite, User Manual. URL http://www.phoronix-test-suite.com/documentation/.
59. The UDOO Platform. URL http://www.udoo.org.
60. The Juno ARM Development Platform. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.boards.juno/index.html.
61. NVIDIA Jetson TK1. URL http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html.
62. Callahan SP, Freire J, Santos E, Scheidegger C, Silva CT, Vo HT. VisTrails : Visualization meets Data Management. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006; 745–747.
63. Apache Storm. URL http://storm.apache.org.
64. Apache Spark. URL http://spark.apache.org.
65. BPF Compiler Collection (BCC). URL https://github.com/iovisor/bcc#tracing.